Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing

Richard Yoo, Christopher Hughes: Intel Labs
Konrad Lai, Ravi Rajwar: Intel Architecture Group
Intel® Transactional Synchronization Extensions (Intel® TSX) can improve HPC application performance

1. Backgrounds on Intel TSX
2. Evaluation of Intel TSX on TM Benchmarks
3. Evaluation of Intel TSX on Real-World Workloads
The Synchronization Problem

- Writing multi-threaded programs
  - Threads concurrently access shared data
  - Conflicting accesses must be synchronized
Coarse-Grained Locks

- Use a lock to guard shared data
  - A thread acquires lock, performs updates, and releases lock
  - Other threads wait
- Downside: serialized all the accesses
Fine-Grained Locks

- Assign a lock for each element
  - Accessing different elements need not serialize
- Such tuning is time consuming and error prone
Can We Do Better?

- What we really want
  - Developer effort of coarse-grained locking
  - Performance of fine-grained locking

- Lock elision
  - Lock-protected critical sections are transactionally executed
  1. Speculatively execute the section without acquiring lock
     - Exposes hidden concurrency
  2. Only enforce serialization when actual data conflicts occur
     - Rollback speculative updates and resume execution
Lock Elision via Transactional Execution

- Exploit property of lock variables
  - Lock variables exhibit temporal silence
  - Lock release restores value of lock prior to lock acquire
- Execute everything atomically (like a single point in time)

```c
If (lock == UNHELD)
lock := HELD
```

**Acquire**

```c
lock := UNHELD
```

**Release**

Speculating thread sees acquired lock
Other threads see a free lock

Temporally Silent-pairs
No need to perform stores
Enabling Lock Elision

- Intel TSX: hardware support to enable lock elision
  - Manages transactional updates
    - Transactional reads are tracked, writes are buffered
    - Other threads cannot observe transactional updates
  - Detects data conflicts
    - On actual data conflict, discard updates and rollback registers
    - Restart execution and acquire lock non transactionally
  - Atomically commit modifications
    - Transactional updates are instantaneously visible
Intel TSX Programming Interface

• Two programming interfaces: HLE and RTM
  – HLE: legacy-compatible, policies baked into hardware
  – RTM: more flexible interface, users can fine-tune

• Instruction Set
  – XBEGIN abort_handler: start transactional execution
  – XEND: commit transactional execution
  – XTEST/XABORT: test or explicitly abort transactional execution

• Some instructions and events may cause aborts
  – Uncommon instructions (syscalls), interrupts, faults, etc
  – Software must guarantee progress w/ non-transactional path
Commercial HTM

• Commercial Implementations
  • 2011 IBM Blue Gene/Q
  • 2012 IBM zEC12 mainframe
  • 2013 Intel 4th generation Core (Haswell)
  • 2014? IBM POWER8
Industrial Considerations

• Provide a clear benefit to customers
  • Improve performance & scalability
  • Ease programmability going forward

• Improve something common and fundamental
  • Widely used critical section/lock-based primitives

• In an easy to use and deploy manner
  • Minimal eco-system impact and effort
  • Clean architectural boundaries

• While managing HW design and validation complexity
<table>
<thead>
<tr>
<th>Conflict Detection</th>
<th>Lazy</th>
<th>Eager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lazy</td>
<td>Optimistic C. Ctrl. DBMS IBM BGQ?</td>
<td>None</td>
</tr>
<tr>
<td>Eager</td>
<td>MIT LTM</td>
<td>Conservative C. Ctrl DBMS MIT UTM LogTM</td>
</tr>
<tr>
<td></td>
<td>Intel/Brown VTM</td>
<td>Intel, IBM BGQ, zEC &amp; Power8 &lt;Azul, Sun Rock&gt;</td>
</tr>
</tbody>
</table>
Features Convergence

• Convergence over basic functionalities...
  • Best effort HTM
  • Leverage cache coherency protocol/cache(s)
  • Strong Isolation
  • Hardware buffering
  • Reasonable buffer size
  • No instruction count limit
  • Checkpoint of Registers
  • Implicitly Transactional

• Some differences...
  • IBM BGQ supports thread speculation
  • IBM zEC supports constrained transactions
  • IBM POWER8 supports supsend/resume
  • IBM zEC/POWER8 supports non-txn stores (but differently).
Using RTM: Lock Elision Success

```c
acquire_lock (mutex);
// execute critical section
release_lock (mutex);
```

**application code**

**Try:**
- `lock xchg mutex, eax`
- `cmp eax, 0`
- `jz Success`

**Spin:**
- `pause`
- `cmp mutex, 1`
- `jz Spin`
- `jmp Try`

**Retry:**
- `xbegin Abort`
- `cmp mutex, 0`
- `jz Success`
- `xabort $0xff`

**Abort:**
- `sub retryCnt, 1`
- `cmp retryCnt, 0`
- `jz acquire_lock`
- `jmp Retry`

**Start txn execution, Abort is fallback**
- Maintain exclusion w/ non-txn
- No non-txn, continue txn execution

**Code example for illustration purpose only**
Using RTM: Lock Elision Success

```c
acquire_lock (mutex);
// execute critical section
release_lock (mutex);
```

Code example for illustration purpose only
Using RTM: Lock Elision Fail

```c
acquire_lock (mutex);
// execute critical section
release_lock (mutex);
```

Application code

- **Try:**
  ```c
  mov eax, 1
  lock xchq mutex, eax
  cmp eax, 0
  jz Success
  ```

- **Spin:**
  ```c
  pause
  cmp mutex, 1
  jz Spin
  jmp Try
  ```

- **Abort:**
  ```c
  xabort $0xff
  ```

- **Retry:**
  ```c
  sub retryCnt, 1
  cmp retryCnt, 0
  jz acquire_lock
  jmp Retry
  ```

Determine whether to retry txn

Fallback to locking

Retry txn

Code example for illustration purpose only
Using RTM: Lock Elision Fail

```
acquire_lock (mutex);
// execute critical section
release_lock (mutex);
```

```
mov eax, 1
Try: lock xchg mutex, eax
    cmp eax, 0
jz Success
Spin: pause
    cmp mutex, 1
jz Spin
    jmp Try
mov mutex, 0
release_lock()
```

**Lock elision can be implemented with RTM**

**Software lock used as the fallback**

Code example for illustration purpose only
Experiment Settings

• Implement lock elision in synchronization library
  – Modified workload macros and pragmas to call our library
  – Optimization: get rid of redundant lock checks

• 4th Generation Core™ Processor with Intel TSX support
  – 4 cores, 2 threads per core
  – L1 data cache (32 KB) buffers transactional updates
  – Conflicts detected at cache line granularity (physical addr)

• Varying range of workloads
  – Microbenchmarks, TM benchmarks, real-world workloads, user-level TCP/IP stack
STAMP Results

- STAMP: benchmark suite extensively used by TM community

- Compare 3 different synchronization schemes
  - **sgl**: critical sections protected through a single global lock
  - **tl2**: software transactional memory (leverages annotations)
  - **tsx**: lock elision w/ Intel TSX
STAMP Results (contd.)

- **Microarchitectural implications on Intel TSX**
  - 32 KB L1 capacity: transactional aborts w/ single thread
  - HyperThreading shares L1 among two threads

- **STAMP covers diverse transactional characteristics**
  - Some workloads stopped critical section refinement at medium/large footprint
  - Real-world workloads less impacted
Real-World Workloads

<table>
<thead>
<tr>
<th>Workload</th>
<th>Description</th>
<th>Sync</th>
</tr>
</thead>
<tbody>
<tr>
<td>graphCluster</td>
<td>Min-cut graph clustering (Kernel 4 of SSCA2)</td>
<td>locks</td>
</tr>
<tr>
<td>ua</td>
<td>Unstructured Adaptive (NPB suite)</td>
<td>atomics</td>
</tr>
<tr>
<td>physicsSolver</td>
<td>Use PSOR to solve force constraints</td>
<td>locks</td>
</tr>
<tr>
<td>nufft</td>
<td>Non-uniform FFT</td>
<td>locks</td>
</tr>
<tr>
<td>histogram</td>
<td>Parallel image histogram</td>
<td>atomics</td>
</tr>
<tr>
<td>canneal</td>
<td>VLSI router (PARSEC)</td>
<td>lock-free</td>
</tr>
</tbody>
</table>

- Computations typically found in the HPC
  - Optimized by domain experts (used to stress Xeon Phi™ processors)

- Apply Intel TSX on top of existing optimizations
  - Locks: simply apply Intel TSX-based lock elision
  - Atomics and lock-free:
    - Convert to regular mem ops, and enclose in lock-protected critical sections
    - Apply Intel TSX-based lock elision
Lockset Elision

• For some workloads, a set of locks need to be acquired to enter a critical section
  – physicsSolver: acquire two locks for each object pair
  – A pattern frequently observed in HPC

• Lock acquisition = costly atomic operations
  – Acquiring a set of locks can impose even higher overheads
  – To amortize, substitute w/ single transactional begin
Transactional Coarsening

- Batch transactions to better amortize overhead
  - *Static* coarsening: merge different critical sections into one
  - *Dynamic* coarsening: combine multiple dynamic instances

  ```c
  #pragma omp atomic
  tmor[ig1] += tx[il1] * third;
  #pragma omp atomic
  tmor[ig2] += tx[il2] * third;
  #pragma omp atomic
  tmor[ig3] += tx[il3] * third;
  #pragma omp atomic
  tmor[ig4] += tx[il4] * third;
  
  ua Code Example
  ```

  ```c
  for (int y = y0; y < y1; y++)
    for (int x = x0; x < x1; x++) {
      if (x % TXNGRAN == 0)
        TM_BEGIN();
      // Update histogram bin
      UPDATE BIN(x);
      if (x % TXNGRAN == TXNGRAN - 1)
        TM_END();
    }
  
  histogram Code Example
  ```

- Tradeoff:
  - Larger transaction = lower overheads, but higher conflict probability
  - Automated tuning approach could be implemented

Code example for illustration purpose only
TSX Evaluation on HPC Workloads

Substitute atomic operations, locks, and non-blocking sync. with RTM
- Average 1.41x speedup with 8 threads

Workloads benefit from RTM by
1. Exploiting concurrency within a critical section (nufft)
2. Reducing the synchronization cost (ssca2, physicsSolver, nufft, histogram)
3. Replacing complex non-blocking sync. w/ regular memory ops (canneal)
Other Results

• RMS-TM
  – Memory allocation and system call within transactional regions
  – Intel TSX provides similar performance to fine-grained lock

• User-Level TCP/IP Stack
  – Modify user-level TCP/IP stack in PARSEC
  – Average 1.31x bandwidth improvement
  – Need conditional variable TM optimization

• Java + Intel TSX
  – Transactionally execute synchronized statements
  – Promising initial results
PARSEC 3 User-Level TCP/IP Stack

- Conditional variable is common in real apps
- Cond_wait always abort
- Busy wait instead
- 1.3x better BW

```c
pthread_mutex_lock(&lock);
while (monitor state not true) {
    // Wait till condition met
    pthread_cond_wait(&lock, &cond);
}
pthread_mutex_unlock(&lock);
```

Listing 4: PThread condition variable wait routine.

```c
pthread_mutex_lock(&lock);
... update monitor state to true ...
// Signal waiting thread
pthread_cond_signal(&cond);
pthread_mutex_unlock(&lock);
```

Listing 5: PThread condition variable signal routine.

```c
while (monitor state not true) {
    // Busy-wait till condition met
    pthread_mutex_unlock(&lock);
    pthread_mutex_lock(&lock);
}
pthread_mutex_unlock(&lock);
```

Listing 6: Busy-wait substitution for conditional wait.

Figure 6: Intel TSX performance on user-level TCP/IP stack. Reports server-side read bandwidth.
Conclusion

• Using Intel TSX for lock elision provides promising results
  – Microbenchmarks, TM benchmarks, real-world workloads, user-level TCP/IP stack

• HPC applications can benefit, too
  – Average 1.41x improvement
  – Key techniques: lockset elision, transactional coarsening
Fun Observations

• Hello World of HTM
  – Atomic { i++; }

• Common Implementation Questions:
  – Size
  – But do you know your need? No

• Common Functional Bugs
  – No fall back code, or retry forever
  – Fall back code and HTM code doesn’t work together

• Common Performance Bugs
  – Lemming Effect. Sustained non-HTM execution
Upcoming Publications

- **HPCA 2014 (TU Dresden, Intel, SAP)**
  Improving In-Memory Database Index Performance with Intel® Transactional Synchronization Extensions
  Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, Wolfgang Lehner

- **PPoPP 2014 (IBM Japan)**
  Eliminating Global Interpreter Locks in Ruby through Hardware Transactional Memory
  *Rei Odaira, Jose Castanos and Hisanobu Tomari*

- **ICDE 2014 (TU Munchen)**
  Exploiting Hardware Transactional Memory in Main-Memory Databases
  Viktor Leis, Alfons Kemper, Thomas Neumann
TSX References

• Intel TSX Specifications
  – “Intel architecture instruction set extensions programming reference.” Chapter 8: Intel transactional synchronization extensions

• Enabling and Optimization Guidelines
  – “Intel 64 and IA-32 architectures optimization reference manual.” Chapter 12: Intel TSX recommendations

• Additional Resources for Intel TSX
  – http://www.intel.com/software/tsx